Generating Code with Polymorphic let
A Ballad of Value Restriction, Copying and Sharing

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Getting polymorphism and effects such as mutation to live together in the same language is a tale worth telling, under the recurring refrain of copying vs. sharing. We add new stanzas to the tale, about the ordeal to generate code with polymorphism and effects, and be sure it type-checks. Generating well-typed–by–construction polymorphic let-expressions is impossible in the Hindley-Milner type system: even the author believed that.

The polymorphic-let generator turns out to exist. We present its derivation and the application for the lightweight implementation of quotation via a novel and unexpectedly simple source-to-source transformation to code-generating combinators.

However, generating let-expressions with polymorphic functions demands more than even the relaxed value restriction can deliver. We need a new deal for let-polymorphism in ML. We conjecture the weaker restriction and implement it in a practically-useful code-generation library. Its formal justification is formulated as the research program.

1 Introduction

This paper revolves around code generation, namely, generating typed, higher-order code for languages such as OCaml. Specifically we deal with one approach to code generation: staging (recalled in §2.2), and the lightweight way of implementing it via code-generating combinators. In our approach, the generated code is assured to be well-formed and well-typed by construction: attempts to produce ill-typed fragments are reported when type-checking the generator itself. In contrast, the post-validation used, for example, in Template Haskell [23], type-checks the code only after it has been completely generated. The errors are thus reported in terms of the generated code rather than the generator, breaking the generator's abstractions[1].

However, staging here is the lens through which to look at the old problem of let-generalization. The unexpected interactions of polymorphism and staging brings into focus the 'too obvious' and hence rarely mentioned assumptions of the value restriction. Generating code that contains polymorphic let-expressions is a non-contrived, real-life application that requires let-generalization of effectful expressions – going beyond what even the relaxed value restriction offers. Staging thus motivates further work on the seemingly closed topic of let-generalization in the presence of effects.

Although program generation is a vast area, surprisingly there has been very little research on typed-assured code generation with polymorphic let. To our knowledge, [15] is the first paper that brings up a staged calculus that has both polymorphism and mutable cells. It is motivated by the unexpected interaction of polymorphism and staging that we describe in §2.3. There are many systems for typed

[1] Post-validation is hence similar to run-time failure of ill-typed code in dynamically-typed languages. However, with a run-time error we can get a stack trace, etc. On the other hand, when post-validating the (typically large and obfuscated) generated code, the generator is long gone and its state can no longer be examined.

Jeremy Yallop and Damien Doligez (Eds.)
ML/OCaml 2015
EPTCS 241, 2017, pp. 1–22, doi:10.4204/EPTCS.241.1

code generation (see [11, §5] for the recent overview) yet polymorphic-let expressions are not included in the target language. The only related, albeit quite remotely, is the work [3] on typed self-interpreters, which does include the representation of polymorphic expressions as code – but lacks any effects. That work is based on System F_ω, which is difficult to use in practice, in part because it lacks type inference. In contrast, in our code generation approach all types are inferred.

Contributions First, the paper presents a new translation from the staged code – with quotations, unquotations and cross-stage persistence – to quotation-free expressions over code-generating combinators. The translation is remarkably simpler than the other unstaging translations. It also translates quoted let-expressions to let-expressions, for the first time giving the chance to generate polymorphic let-expressions, well-typed by construction. Second, we present the first library of typed code combinators whose target language includes polymorphic let-bindings and effects. The library requires no first-class polymorphism, no type annotations and, combined with the unstaging translation, is suitable for implementing staging by source-to-source translation to combinators. The library solves the problem that the author claimed in 2013 to be unsolvable [13].

Although the translation and the library are already practically useful, their formalization requires deeper understanding of polymorphism and effects. The paper proposes a research program, which will have to open the old value-restriction wounds and could finally heal them. Thus we end up posing more questions – the questions that could not have been asked before.

The paper starts with extensive background. §2.1 recalls let-polymorphism and the ways to restrict it in the presence of effects. That section describes the copying-sharing dichotomy that reverberates through the rest of the paper. §2.2 introduces staging, using MetaOCaml as an example, and §2.3 describes the unexpected interactions of staging and polymorphism. We then describe in §3 the novel translation that systematically eliminates quoted expressions, replacing them with applications of code combinators. Alas, the translation does not seem work for polymorphic let-expression, as shown in §3.2. It can be made to work; §4 explains how. As in the ordinary ML, reference cells and polymorphism is a dangerous mix; our translation hence needs some sort of a restriction, weaker than even the relaxed value restriction. §4.1 discusses the solutions and the many follow-up problems.

We will be using OCaml throughout the paper for concreteness. However the discussion equally applies to any other typed, higher-order language with polymorphism and effects.

The complete code accompanying the paper is available at
`http://okmij.org/ftp/meta-programming/polylet.ml`

2 Background

This background section recalls let-generalization; its problems in the presence of effects; staging; and the unexpected interaction of generalization and staging that calls up the assumptions of the value restriction. The section introduces the running examples used later in the paper.

2.1 Let-polymorphism

Since the early days of LISP and ISWIM [16], let-expressions let us introduce and name common or notable expressions which are used, often repetitively, later in the code. Here is a simple example:

```
(1)  let x = [1] in
     (2:: x ,3:: x)
```

It may be regarded as a sort of a ‘macro’ that expands into

```
(2)    (2::[1],3::[1])
```

In fact, such a ‘macro-expansion’ – copying (inlining) the let-bound expression into the places marked by the let-bound variable – is the meaning given to let-expressions in Landin’s ISWIM [16]. The alternative to this copying, or substitution-based semantics is sharing. It views (1) as introducing the expression [1] that is shared across all occurrences of x. Hence the two lists in (2) share the common tail. Copying vs. sharing reverberates throughout the paper as the constant refrain. If our program simply prints out (2), the two semantics are indistinguishable. The equivalence lets the compiler choose inlining or sharing as fits.

Likewise, the code

```
(3)   let x = [] in
      (2:: x,"3" ::x)
```

may be viewed as a macro that expands into

```
(4)    (2::[], "3" ::[])
```

It is tempting to also regard (3) as the sharing of the empty list across the two components of the returned pair. Unlike (2), however, [] in (4) has different types: namely, int list in the first component vs. string list in the second. Thus comes the problem of what type to give to the shared value and to x.

The answer developed by Milner [20] was polymorphism: the same expression that occurs in – has been copied into – differently typed contexts may be shared and given the common, the most general, polymorphic type (see also the extended discussion in [5]). The empty list [] has the type α list to be fully determined by the context; α is the placeholder: a (unique) type variable. In (4), the contexts determine the type as int list and string list, respectively. In (3), the context of the right-hand side (RHS) of the let-binding has not determined what α list should be. In that case, the type is *generalized* to the polymorphic type schema: $\forall\alpha.\ \alpha$ list.

Formally, the typing of let-expressions is represented by the (GenLet) rule below. The rule is written in terms of the judgments $\Gamma \vdash \mathsf{e} : \mathsf{t}$ that an expression e has the type t in the type environment Γ (which lists the free variables of e and their types).

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{t} \qquad \Gamma, \mathsf{x} : GEN(\Gamma, \mathsf{t}) \vdash \mathsf{e}' : \mathsf{t}'}{\Gamma \vdash \textbf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}' : \mathsf{t}'}\ \mathit{GenLet} \qquad \frac{\mathsf{x} : \forall\alpha_1 \ldots \alpha_n.\mathsf{t} \in \Gamma}{\Gamma \vdash \mathsf{x} : t\{\alpha_1 = \mathsf{t}_1 \ldots \alpha_n = \mathsf{t}_n\}}\ \mathit{Inst}$$

The generalization function $GEN(\Gamma, \mathsf{t})$ for the type t with respect to the type environment Γ quantifies the free type variables of t that do not occur as free in Γ:

$$GEN(\Gamma, \mathsf{t}) = \forall\alpha_1 \ldots \alpha_n.\mathsf{t} \qquad \text{where}\{\alpha_1 \ldots \alpha_n\} = FV(\mathsf{t}) - FV(\Gamma)$$

where $FV(\cdot)$ denotes the set of free variables. When a variable with the polymorphic type schema such as x: $\forall\ \alpha.\ \alpha$ list in (3) is used in an expression, e.g., 3::x, the schema is converted to a more specific type, int list in our example: see the rule (Inst). The underlying assumption is that the value named by x indeed has the same representation for all instances of the polymorphic type schema and hence may be shared, even across differently-typed contexts; the instantiation is a purely type-checking-time operation that behaves like identity at run-time. One may say that the motivation of polymorphism is to extend the equivalence of the copying and sharing semantics to the cases like (3).

Side-effects break the equivalence of copying and sharing.

```
(5)   let x = begin printf "bound"; [1] end in
      (2:: x ,3:: x)
```

If the right-hand side is copied, substituting for the occurrences of x in (5), the string "bound" is printed twice. If the RHS is first reduced, however, (5) turns to the earlier (1), where x is bound to the value that can be either shared or copied. Hence the copying/sharing equivalence holds even in the case of (5), if we regard variables as bound to values – as we do in call-by-value languages[2]. The polymorphic case should likewise be unproblematic:

```
(6)  let x = begin printf "bound"; [] end in
     (2::x,"3" ::x)
```

The polymorphic equality of OCaml can distinguish sharing and copying:

```
(7)  let x = [1] in
     x == x
```

whereas (7) returns the result **true**, the expression [1] == [1] produces **false**. Another, universal way to distinguish sharing and copying uses mutable data structures, in particular, mutable cells [1]. Let's define

```
(8)  let rset : α list ref → α → α list = fun r v →
       let vs' = v :: !r in
       r := vs';
       vs'
```

that prepends the value v to the list stored in the reference cell r, stores the new list in the cell and returns it. Then

```
(9)  let x = ref [1] in
     (rset x 2, rset x 3)
     ⇝ ([2; 3; 1], [3; 1])
     (rset (ref [1]) 2, rset (ref [1]) 3)
     ⇝ ([2; 1], [3; 1])
```

produce the different results as shown underneath the expressions. Since the distinction between copying and sharing is generally visible, there is no longer freedom of choosing between the two. OCaml uses sharing for **let**-expressions, performing inlining (copying) only when it can see the equivalence.

The example paralleling (3) however does not type-check.

```
(10)  let x = ref [] in
      (rset x 2, rset x "3")
      (* Does not type-check! *)
```

As we have just seen, with reference cells, sharing and copying differ and the OCaml compiler has to use the default sharing. Had the expression type-checked, at run-time rset x "3" would modify the empty list stored in x by prepending the string "3" to it. The expression rset x 2 will then try to prepend the integer 2 to the contents of x, which by that time is the string list ["3"]. Clearly that is a program that has "gone wrong". We should well remember this example: we shall be seeing it, in different guises, all throughout the paper.

Although the RHS of the let-binding in (10) has the type α list **ref** with the type variable that could be generalized, it should not be, to prevent (10) from type-checking. Intuitively, sharing and copying of a reference cell have different semantics, hence it should not get the polymorphic type schema.

The danger of giving reference cells a polymorphic type has been recognized early on [26]. So has the problem of how to restrict (GenLet) from applying to "dangerous" expressions. The most straightforward solution, used in the early ML and OCaml for a long time, was to limit reference cells to base types

[2]Another way to restore the equivalence is to regard x as bound to an expression that is evaluated only at the places of x's occurrence. That was the idea of Leroy's call-by-name polymorphism [18].

only. The restriction made it impossible however to write any polymorphic function that internally uses reference cells. A good overview of less draconian approaches is given in [8]. The most widely implemented, because of its balance of expressiveness with sheer simplicity, is the value restriction [27]: applying (GenLet) only to those let-expressions whose RHS is syntactically a value. Since **ref** [] in (10) is not a value, x is not generalized and its occurrences in differently typed contexts will raise the type error. On the other hand, [] in (3) is a value and x there does get the polymorphic type. Strictly speaking, x in (6) should not be generalized either. However, it is syntactically obvious that the printing effect has no contribution to the result of the containing expression. The RHS of (6) is what is called 'non-expansive'. OCaml generalizes non-expansive expressions, not just values.

Although the value restriction on balance proved expressive enough for many ML programs, as OCaml gained features such as objects, polymorphic variants and a form of first-class polymorphism (enough to support polymorphic recursion), the restrictiveness of the value restriction was felt more and more acutely. Against this backdrop, Garrigue introduced the 'relaxed value restriction' [8], which we briefly overview below as we will be relying on it.

The relaxed value restriction explores the close analogy between type instantiation and subtyping. It can also be justified from the point of view of copying-sharing: a value occurring in differently-typed contexts may be let-bound and shared if it can be given the 'common type', the supertype of the types expected by the contexts. The coercion to a subtype, like the type instantiation, is a compile-time–only operation, behaving as identity at run-time. Suppose a value has the type zero c where zero is the empty type, and it can be *coerced* by subtyping to the type t c for any t. We may as well then give the variable that is let-bound to the value the type $\forall\alpha.\ \alpha$ c, which can then be *instantiated* to t c. Since zero is (vacuously) coercible to any type, a value of the type zero c can be coerced to t c only when the type zero occurs covariantly in zero c. Hence the relaxed value restriction: If the expression e in **let** x = e **in** e′ has a type with covariant type variables (which do not occur in the typing context), they are generalized in the type inferred for x. (The actual implementation is somewhat more restrictive: see [8] for details.)

For example, x below is generalized

```
(11)  let x = let r = ref [] in !r in
      (2:: x,"3" ::x)
```

despite the fact the RHS is an expression – moreover, the expression whose result comes right from a reference cell. Still, the result has the type α list whose type variable is covariant (with List.map being the witness of it). On the other hand, the type of reference cells α **ref** is non-variant and hence x in (10) remains ungeneralized. The relaxed value restriction applies not only to built-in data types but also to user-defined and abstract ones:

```
(12)  type +α mylist = List of α list
      let mklist : α list → α mylist = fun x → List x
      let mycons : α → α mylist → α mylist =
         fun x → function List l → mklist (x :: l)
      let x = mklist [] in
      (mycons 2 x, mycons "3" x)
```

Although the RHS of the let-binding is an expression, x is generalized because the type α mylist is covariant in α. It is declared to be covariant, by the $+\alpha$ covariance annotation. The compiler will check that the RHS of the type declaration really uses the type variable α covariantly. The compiler can also infer the variance, hence the annotations are normally omitted. They are necessary only for abstract types, whose declaration lacks the RHS.

Overall, the relaxed value restriction turned up even better balanced, accommodating not just polymorphic functions but polymorphic data (including row-polymorphic data such as extensible records and variants), whose construction often involves computations. The relaxed value restriction was almost enough for implementing staging via code combinators – but not quite, as we see in §4.1. Let us first review staging.

2.2 Staging

Staging is an approach to writing programs that generate programs; it may be regarded as a principled version of Lisp quotation. For example, MetaOCaml lets us quote any OCaml expression, by enclosing it within .< and >. brackets:

```
(1)   let c = .<1 + 2>.
      ⇝ val c : int code = .<1 + 2>.
```

A quoted expression, representing the generated code, is the value of the type α code, and can be bound, passed around, printed – as well as saved to a file and afterwards compiled or evaluated. For that reason, the code value such as c is called a 'future-stage' expression (or, an expression at level 1), to contrast to the code that is being evaluated now, at the present, or 0, stage. An expression that evaluates to a code value can be spliced-in (or, unquoted, in Lisp terminology) into a bigger quotation:

```
(2)   let cb = .<fun x → .~c + x>.
      ⇝ val cb : (int → int) code = .<fun x_1 → (1 + 2) + x_1>.
```

The spliced-in expression is marked with .~, which is called an escape. The generated code can be executed by the function run (in the module Runcode), reminiscent of Lisp's eval:

```
(3)   open Runcode
      val run :: α code → α

      let cbr = run cb
      ⇝ val cbr : int → int = <fun>
      cbr 2
      ⇝ − : int = 5
```

Running cb hence compiled the cb code of a function into an int→int function that can be invoked at the present level. As one expects, running the code indeed invokes the compiler and the dynamic linker. The run operation hence lets us generate code at run-time and then use it – in other words, it offers run-time code specialization.

When generating functions it is natural to require that the behavior of the resulting program should not depend on the choice of names for bound variables. For example,

```
(4)   let c1 = .<fun x → .~(let body = .<x>. in .<fun x → .~body>.)>.
      ⇝ val c1 : (α → β → α) code = .<fun x_1 → fun x_2 → x_1>.

      let c2 = .<fun y → .~(let body = .<y>. in .<fun x → .~body>.)>.
      ⇝ val c2 : (α → β → α) code = .<fun y_3 → fun x_4 → y_3>.
```

The expressions c1 and c2 should build the code that behaves the same when evaluated. This is indeed the case, as one can see from the generated code, printed underneath. If we write this example with quotations in Lisp, the expressions are no longer equivalent: whereas c2 generates the code for the K combinator, c1 builds a function that takes two arguments returning the second one. Lisp quotations are hence not hygienic.

When generating code for a typed language, it is also natural to require that the produced code is type-correct by construction. For that reason, the code type is parametrized by the type of the generated expression, as we saw for c, cb, etc. The formal treatment of type soundness is well covered in [24, 4] and will be briefly reminded of in §2.3.

MetaOCaml has yet another facility, which has no special syntax and is easy to overlook. Let us look again at .<1 + 2>. and ponder the addition operation there. In the ordinary OCaml expression 1+2, the addition is the ordinary function, defined in the Pervasives module. The addition in .<1+2>. refers to exactly the same function: MetaOCaml permits any value of the generator to appear in the generated code. This is called "cross-stage persistence" (CSP) (see [24] for more discussion). One may think of CSP identifiers as references to 'external libraries'.

The trivial code for the addition of two numbers has already demonstrated how wide-spread CSP is. Let us show a more explicit example of CSP, brought about by the function

```
(5)  let lift : α → α code = fun x → .<x>.
```

The following example then produces the code as shown (compare with (2)):

```
(6)  .<fun x → .~( lift (1 + 2)) + x>.
     ⇝  − : (int → int) code = .<fun x_1 → (* CSP x *) Obj.magic 3 + x_1>.
```

In contrast to (2), here the addition of (1+2) is done at the code generation time; the generated code includes the computed value. CSP hence lets us do some of the future-stage computations at the present stage, and hence generate more efficient code. The bizarre Obj.magic appearing in the generated code is the artifact of printing. The following code

```
(7)  .<fun x → .~(let y = 1 + 2 in .<y>.) + x>.
     ⇝  − : (int → int) code = .<fun x_2 → 3 + x_2>.
```

(where the CSP identifier y is known to be of the int type) produces the more obvious result.

Our refrain of copying vs. sharing repeats for CSP. When a value from the present stage is used at a future stage, do the two stages share the value or does the future stage get a separate copy? Unfortunately this issue is not discussed in the literature let alone formally addressed[3] – which is a pity since it is responsible for the unexpected soundness problem to be described in the next section. The case of a global (library) identifier seems clear: code such as .<succ 3>. contains the identifier succ that refers to the same library function it does at the present level. Whether that function is shared or copied between the present-stage and the generated code depends on the inlining strategy of the compiler and the static vs. dynamic linking. One could expect sharing/copying to be equivalent in this case.

The cross-stage persistence of a locally-created value is much less clear[4]:

```
(8)  let cs =
       let z = string_of_float @@ Sys.time () in
       .< print_endline z>.
     ⇝  val cs : unit code = .<Pervasives. print_endline "0.051">.
```

One may imagine that the code value .<print_endline z>. (represented, say, as an AST) contains the pointer to a string allocated in the running-program heap – the same pointer that is denoted by the local variable z. Then run cs will print the value of that string on the heap. The present and the future stage hence share the string. Rather than running cs however, we may save it to a file, as library code for use in other programs. In this case, when the generated code is evaluated the generator program is long gone, along with its heap. Therefore, when storing a code value to a file we must serialize all its CSP values, creating copies. In the upshot, cross-staged persistent library identifiers are always shared; other CSP values are shared if the code value is run, and copied otherwise. The semantics of CSP is indeed

[3]the exception being the work [15] which was inspired by the problem we discuss in §2.3.

[4]The right-associative infix operator @@ of low precedence is application: f @@ x + 1 is the same as f (x + 1) but avoids the parentheses. The operator is the analogue of $ in Haskell.

$$\frac{\Gamma \vdash^n \mathsf{e : t'{\to}t} \quad \Gamma \vdash^n \mathsf{e' : t'}}{\Gamma \vdash^n \mathsf{e\ e' : t}} \qquad \frac{\Gamma, \mathsf{x}^n : \mathsf{t'} \vdash^n \mathsf{e : t}}{\Gamma \vdash^n \mathbf{fun}\ \mathsf{x \to e : t'{\to}t}}$$

$$\frac{\Gamma \vdash^n \mathsf{v : t} \qquad \Gamma, \mathsf{x}^n : GEN(\Gamma, \mathsf{t}) \vdash^n \mathsf{e' : t'}}{\Gamma \vdash^n \mathbf{let}\ \mathsf{x = v\ in\ e' : t'}}\ GenLet$$

$$\frac{\Gamma \vdash^{n+1} \mathsf{e : t}}{\Gamma \vdash^n \mathsf{<e> : t\ code}}\ Bracket \qquad \frac{\Gamma \vdash^n \mathsf{e : t\ code}}{\Gamma \vdash^{n+1} \mathsf{\tilde{}\ e : t}}\ Escape \qquad \frac{\Gamma \vdash^n \mathsf{x : t}}{\Gamma \vdash^{n+1} \mathsf{x : t}}\ CSP$$

Figure 1: Type system of a staged language

intricate. We have just described the CSP implementation in the extant MetaOCaml; there is an ongoing discussion of it and its possible improvements [5].

The question of sharing vs. copying CSP becomes non-trivial when the CSP value is mutable:

```
(9)   let r = ref 0 in
      let cr = .<incr r>. in
      run cr; run cr; !r
     ⇝ − : int = 2
```

Mutable CSP values naturally arise when run-time specializing imperative code. They can be used for cross-stage communication, e.g., counting how many times the code is run, as shown in (9) – which works as intended only with the shared CSP. Sharing of mutable CSP values is also responsible for the unexpected problem with let-polymorphism, detailed next.

2.3 Let-polymorphism and Staging

For a long time let-polymorphism and staging were considered orthogonal features. It is not until 2009 that their surprising interaction was discovered[6]; it has not been formally published. Before describing this interaction, we first briefly remind the type system of a staged language, on a representative subset of MetaOCaml.

Staging adds to the base language the expression forms for brackets <e> and escapes ˜ e and the type of code values t code. We use the meta-variable x for variables, e for expressions, v for values, and t for types. The type system is essentially the standard, Figure 1. It is derived from the type system of [24] by replacing the sequence of no-longer used classifiers with the single number, the stage level. (Since brackets may nest, there may be an arbitrary number of future stages.) The judgments have the form $\Gamma \vdash^n \mathsf{e : t}$: they are now indexed by the level of the expression; the level is incremented when type-checking the expression within brackets and decremented for escapes. The identifiers within the typing environment Γ are now indexed by the level at which they are bound. The (GenLet) rule reflects the value restriction.

Staging thus contributes the three rules (Bracket), (Escape) and (CSP) and the indexing of the environment and the judgments by the stage level. If the program has no brackets, the stage level stays at 0 and the type system degenerates to the one for the (subset of the) ordinary OCaml. Moreover, except for the three staging-specific rules, the rest are the ordinary OCaml typing rules, uniformly indexed by the stage level. Thus, aside from brackets, escapes and CSP, the type-checking of the staged code proceeds

[5] http://okmij.org/ftp/ML/MetaOCaml.html#CSP

[6] http://okmij.org/ftp/meta-programming/calculi.html#staged-poly

identically to that for the ordinary code. In particular, let-expressions within brackets are handled and generalized the same way they do outside brackets. For example:

```
(1)   .<let x =  [] in  (2:: x,"3" ::x)>.
      .<let f =  fun x → x in (f  2,  f  "3")>.


(2)   .<let x =  ref []  in  ( rset  x  2,  rset  x "3")>.  (* Does not type−check! *)
```

It appears hence that let-generalization and staging are orthogonal features.

Consider however the following code

```
(3)   .<let f =  fun () → ref  []  in
           ( rset  (f  ())  2,  rset  (f  ())  "3")
      >.
```

The type-checker accepts it and infers the type (int list ∗ string list) code. The variable f hence gets the polymorphic type. After all, the RHS of the let-binding is syntactically the (functional) value. There is really nothing wrong with (3): f can be either copied or shared across its uses without the change in semantics: the invocation f () in either case will produce a fresh reference cell holding an empty list, later modified by prepending either 2 or "3" to its contents.

Now consider the simple modification, along the lines of (6) in §2.2:

```
(4)   let  cbad =
      .<let f =  fun () → .~( lift  (ref  [])) in
           ( rset  (f  ())  2,  rset  (f  ())  "3")
      >.
      run  cbad
      ⇝  Segmentation  fault
```

It is also accepted, with the same inferred type – for any version of MetaOCaml including the current one. The RHS of the let-binding is still syntactically a function; we merely modified its body. Running that code however ends in the segmentation fault. One should not be surprised: we have managed to generate and type-check (10) from §2.1, the canonical example of the unsoundness of polymorphism for reference cells.

Thus staging breaks the restriction of the value restriction, unleashing the unsound generalization. If we re-examine the value restriction we now notice an assumption, which is rarely stated explicitly: there are no literals of reference types; every expression of the type t **ref** is not syntactically a value. Cross-stage persistence, however, lets one stage share its values with a future one. Suddenly there are literals of the reference types: these are values imported from the generator into the generated code.

The problem has been overlooked for more than a decade because none of the formalizations of staging have been complete enough, and hence do not handle let-polymorphism along with reference cells and shared CSP. There is currently no fix for the unsound let-generalization problem. One solution is proposed in [15] but it is restrictive. Another possible solution is to force the CSP locally-created values to follow the copying semantics. One may also prohibit generalization if the RHS of a future-stage let-binding contains an escape, thus introducing the explicit correlation of staging and let-polymorphism. Along with bad programs, all these proposals outlaw good ones. Investigating these trade-offs and finding better ones is the subject of future work. The present paper does not solve the unsound staged let-generalization problem either. However, we build a simpler framework to deal with it, reducing the problem to non-staged generalization.

3 Translating the Staging Away

The stymieing problems encountered in the previous section come at the confluence of staging, let-polymorphism and effects. It is only natural to wish to investigate them in a simpler setting; for example, to find a way to translate a staged calculus into the ordinary one. There have been indeed proposed several 'unstaging translations' [28, 9, 7], with similar motivations.

Translating the staging away is also practically significant, as the method for implementing staged languages. The most attractive is a source-to-source translation: it lets us implement MetaOCaml just as a pre-processor to OCaml, fully reusing the existing OCaml compiler without modifying it (and having to bear the burden of maintaining the modifications, in sync with the mainline OCaml). This practical application is the main reason to be interested in unstaging translations.

Unfortunately, none of the existing unstaging translations deal with polymorphic let-expressions. Furthermore, an attempt to add them, described in §3.2, requires first-class polymorphism, making the translation unworkable as a source-to-source translation. Despite its attractiveness, the approach is a dead-end – as has been widely acknowledged, including by the author.

We first describe in §3.1 how well the translation approach works without the polymorphic let, before illustrating how it does not with it. §4 introduces the solution along with the new questions it poses for let-polymorphism.

3.1 Staging via Code Combinators

The simplest approach for adding quotation to an existing language is to write a pre-processor that translates quoted expressions into ordinary ones, which use pre-defined functions that build and combine code values, so-called code combinators [25, 28, 22]. Code combinators may of course be used for code generation directly, rather than through quotation, as has been well demonstrated in Scala [22]. That said, we will explain code combinators in the context of an unstaging translation, from the language with quotations to the language without them – motivated by the practical benefits of such translation.

Our source language, Figure 2, is a simple subset of MetaOCaml (for now, without let-expressions). From now on, we restrict staging to two-levels only – in other words, considering brackets without nesting – as this turns out the overwhelmingly common use of staged languages. The constants of the language are integer i and string s literals and the empty list. Besides abstraction and application the language includes pairs, consing to a list and the creation and dereference of reference cells. We take the mutation function rset: α list **ref** $\rightarrow \alpha \rightarrow \alpha$ list defined in §2.1 as a primitive. Cross-stage persistent library identifiers such as $+$ are worked out into the syntax. On the other hand, cross-stage persistence of other identifiers must be explicitly marked with the % syntax. (The marking is inferred in MetaOCaml.)

Constants	c ::=	i | s | []
Variables	x,y,z,f	
Expressions	e ::=	x | c | e e | **fun** x → e | e + e | (e,e) | e :: e | **ref** e | !r | rset r
Staged expressions	e +::=	.<e>. | .~e | %x

Figure 2: Source and target languages for the unstaging translation

The target language of the translation is OCaml, without 'Meta', i.e., without the staged expressions. On the other hand, it has additional constants for code generation, defined by the following signature

```
(1)  module type Code = sig
       type +α cod
```

Translation at the present-stage ⌊ e ⌋

⌊ x ⌋	↦	x	⌊ **fun** x → e ⌋	↦	**fun** x → ⌊ e ⌋
⌊ c ⌋	↦	c		...	
⌊ e1 e2 ⌋	↦	⌊ e1 ⌋ ⌊ e2 ⌋	⌊ .<e>. ⌋	↦	⌈ e ⌉

Translation at the future-stage ⌈ e ⌉

⌈ x ⌉	↦	x	⌈ **ref** e ⌉	↦	ref_ ⌈ e ⌉
⌈ i ⌉	↦	int i	⌈ !e ⌉	↦	rget ⌈ e ⌉
⌈ s ⌉	↦	str i	⌈ rset e ⌉	↦	rset ⌈ e ⌉
⌈ [] ⌉	↦	nil	⌈ e1 e2 ⌉	↦	app ⌈ e1 ⌉ ⌈ e2 ⌉
⌈ e1 + e2 ⌉	↦	add ⌈ e1 ⌉ ⌈ e2 ⌉	⌈ **fun** x → e ⌉	↦	lam (**fun** x → ⌈ e ⌉)
⌈ (e1,e2) ⌉	↦	pair ⌈ e1 ⌉ ⌈ e2 ⌉	⌈ .~e ⌉	↦	⌊ e ⌋
⌈ e1 :: e2 ⌉	↦	cons ⌈ e1 ⌉ ⌈ e2 ⌉	⌈ %x ⌉	↦	csp x

Figure 3: Unstaging translation

```
  val int :  int → int cod
  val str :  string → string cod
  val add:   int cod → int cod → int cod
  val lam:   (α cod → β cod) → (α→β) cod
  val app:   (α→β) cod → (α cod → β cod)

  val pair : α cod → β cod → (α ∗ β) cod
  val nil :  α list cod
  val cons:  α cod → α list cod → α list cod

  val ref_ :  α cod → α ref cod
  val rget :  α ref cod → α cod
  val rset :  α list ref cod → α cod → α list cod

  val csp:    α → α cod (∗ CSP local values ∗)
end
```

The signature specifies the collection of typed combinators to generate code for our subset of OCaml: int 1 builds the literal 1 code, add combines two pieces of code into the addition expression, etc. The combinator lam builds the code of a function; its argument is an OCaml function that returns the code for the body upon receiving the code for the bound variable. A MetaOCaml expression like

```
(2)  fun x → .<fun y → (y + 1) :: .~x>.
```

then corresponds to the plain OCaml expression with the code combinators:

```
(3)  fun x → lam (fun y → cons (add y (int 1)) x)
```

Formally the unstaging translation is specified in Figure 3, with two sets of mutually recursive rules: ⌊ e ⌋ deals with the present-stage expressions of the source language and ⌈ e ⌉ handles expressions within brackets. The former is essentially identity, with the single non-trivial rule for brackets. The translation seems straightforward, which is a great surprise since the related unstaging translations [6, §3] and [9, 7] are all excruciatingly more complex and type-directed. The shown translation is novel, which will become apparent as we discuss the implementation of the Code signature later.

Our translation is clearly syntax-directed but not type-directed. Hence it is a source-to-source translation, which can be done by a macro-processor such as camlp4 or a stand-alone pre-processor. The rest

$$\begin{array}{lllll} \lceil \mathsf{t\ code} \rceil \mapsto \lceil \mathsf{t} \rceil\ \mathsf{cod} & \lceil \mathsf{x}^0 \mathsf{: t} \rceil & \mapsto & \mathsf{x}^0\mathsf{: t} & \lceil \Gamma \vdash^0 \mathsf{e : t} \rceil \mapsto \lceil \Gamma \rceil \vdash^0 \lfloor \mathsf{e} \rfloor : \lceil \mathsf{t} \rceil \\ \quad \text{identity otherwise} & \lceil \mathsf{x}^1 \mathsf{: t} \rceil & \mapsto & \mathsf{x}^0\mathsf{: t\ cod} & \lceil \Gamma \vdash^1 \mathsf{e : t} \rceil \mapsto \lceil \Gamma \rceil \vdash^0 \lceil \mathsf{e} \rceil : \lceil \mathsf{t} \rceil \mathsf{cod} \end{array}$$

Figure 4: Translation for types, typing environments and judgments

of the language system (type-checking, code-generation, standard and user-defined libraries) is used as it is.

The second property of the translation is that bindings within brackets are translated to ordinary lambda-bindings. Coupled with the appropriate implementation of the lam combinator, this property makes it easy to ensure hygiene. Correspondingly, variables bound within brackets are translated to the ordinary, present-stage variables – with the change in type from t to t cod. One can see that change from the type of lam, and more clearly from Figure 4, which extends the translation to the typing judgments and environments described Figure 1. The translation is typing-preserving:

Proposition 1 *If* $\Gamma \vdash^n e : t$ *holds then* $\lceil \Gamma \vdash^n e : t \rceil$ *holds as well*

In other words, a well-typed two-stage MetaOCaml expression is translated into a well-typed OCaml expression. The proposition is easily proven by induction on the typing derivation. If we also ensure that individual code combinators produce well-typed code (see below), any typing errors in the quoted code manifest themselves as OCaml type errors emitted when type-checking the translated expression. Absent such errors, the quoted expression, and hence the generated code, are type-correct.

The following figure shows two implementations of the Code signature. CodeString combinators generate ML code as text strings, justifying their name ‘code-generating combinators’.

```
(4)  module CodeString = struct
       type α cod = string

       let int    = string_of_int
       let str x = "\"" ^ String.escaped x ^ "\""
       let add x y = paren @@ x ^ "␣+␣" ^ y
       let lam body =
         let var = gensym "x" in
         "fun␣" ^ var ^ "␣→␣" ^ body var
       let app f x = paren @@ f ^ "␣" ^ x
       ...
       let csp x = ... marshaling/unmarshaling ...
     end
```

CodeReal is a meta-circular interpreter, representing a code value as an OCaml thunk (which is also a value).

```
(5)  module CodeReal = struct
       type α cod = unit → α
       open DynBindRef

       let int x = fun () → x
       let str x = fun () → x
       let add x y = fun () → x () + y ()
       let lam body =
         let r = dnew () in
         let b = body (fun () → dref r) in
         fun () →
           let denv = denv_get () in
           fun x → dlet denv r x b
       let app f x = fun () → f () (x ())
```

```
    let  nil  =  fun () → []
    . . .
    let csp x =  fun () → x
  end
```

The code is utterly trivial, with the exception of lam, which does what a closure has to do: capture the environment at the point of its creation. We rely on the simple interface for dynamic binding:

```
(6)  module type DynBind =  sig
       type α dref
       type denv
       val dnew: unit  → α dref
       val dref:  α dref  → α
       val dlet :  denv → α dref  → α → (unit  → ω) → ω
       val denv_get:  unit  → denv
     end
```

where dnew creates a new unbound variable, dref dereferences it, denv_get captures the current environment and dlet denv r x body sets the current environment to denv, binds r to x in it and evaluates the body, whose result is returned after the original environment is restored. The implementation, using either reference cells or delimited control is straightforward; see the accompanying source code for details. The source code contains more examples of the staged translation, including the obligatory factorial: although our Code interface offers neither conditional branching nor recursive bindings (nor multiplication, for that matter), they are all obtainable via CSP.

Proposition 2 *If* e : t code *is a program in our subset of MetaOCaml, then* $\lfloor$ e $\rfloor$: unit→t *is the plain OCaml program (assuming the* Code *interface is implemented by* CodeReal*) such that* run e *is observationally equivalent to* $\lfloor$ e $\rfloor$ *().*

Although the intuitions are clear, the rigorous proof of this proposition is a serious and interesting task. We leave the proof as a PhD topic. The proposition justifies the name ‘unstaging translation’: translating staged OCaml code to plain OCaml. Our translation is remarkably simple because of the novel implementation of lam in CodeReal. The earlier translations had to explicitly represent and translate the typing and the value environments of an expression. DynBind lets us piggy-back on the typing environment of OCaml.

One can also intuitively see that CodeString and CodeReal correspond: the behavior of the code produced by CodeString is the same as the behavior of running the thunk of CodeReal (modulo the difference in the copying/sharing semantics of CSP). The OCaml type-checker ensures that any thunk built by CodeReal combinators is well-typed; therefore, it “will not go wrong” thanks to the soundness of OCaml. Hence the code generated by CodeString will also be well-typed and will not go wrong either. The existence of the CodeReal implementation is thus crucial to assuring the soundness of code generation. Yet another proof of soundness is obtained through another implementation of Code, back into MetaOCaml:

```
(7)  module CodeCode =  struct
       type α cod =  α code

       let  int  (x: int )     =  .<x>.
       let  str  (x: string ) =  .<x>.
       let  add x y =  .<.~x + .~y>.
       let  lam body =  .<fun x → .~(body .<x>.)>.
       let  app x y =  .<.~x  .~y>.
       . . .
       let  csp x =  .<x>.
     end
```

Proposition 3 *If* $\mathsf{e} : \mathsf{t}\ \mathsf{code}$ *is a program in our subset of MetaOCaml, then* $\lfloor \mathsf{e} \rfloor : \mathsf{t}\ \mathsf{code}$ *is the equivalent MetaOCaml program (assuming the* Code *interface is implemented by* CodeCode*): that is,* e *and* $\lfloor \mathsf{e} \rfloor$ *have the same side effects and either both diverge, or return identical (modulo* α*-conversion) code values.*

The proof is left as another PhD topic.

Implementing staging by the translation into code combinators works surprisingly well: Scala's Lightweight Modular Staging (LMS) is based on similar ideas [22]. Scheme's implementation of quasiquote is also quite alike; only it pays no attention to quoted bindings and is hence non-hygienic. The translation becomes more complex as we add to the target language more special forms such as loops, pattern matching, type annotations, etc. They pose problems, but they can and have been dealt with, e.g., in [22]. What could not be dealt with is let-polymorphism.

3.2 The Let-Polymorphism Problem

The staging translation runs into the roadblock once we add polymorphic let-bindings, to handle expressions such as those shown in §2.3, repeated for reference below.

```
(1)   .<let x = [] in (2::x,"3"::x)>.

(2)   .<let f = fun x → x in (f 2, f "3")>.
```

It may seem we merely need to add to the Code signature the combinator that combines app and lam:

```
(3)   val let_ : α cod → (α cod →β cod) → β cod
```

and the corresponding translation rule

$$\lceil \textbf{let}\ \mathsf{x} = \mathsf{e1}\ \textbf{in}\ \mathsf{e2} \rceil \quad \mapsto \quad \mathsf{let\_}\ \lceil \mathsf{e1} \rceil\ (\textbf{fun}\ \mathsf{x} \to \lceil \mathsf{e2} \rceil)$$

analogous to lam. Then (1) is translated to

```
(4)   let_ nil (fun x → pair (cons (int 2) x) (cons (str "3") x))
```

which, unfortunately, does not type-check.

Recall that our unstaging translation maps bindings in the quoted code to ordinary lambda-bindings. This exactly is the problem: unlike let-bindings, lambda-bindings in ML are not generalizable. First-class polymorphism, if available, does not help since it requires type annotations, which preclude the source-to-source translation, done before type checking.

Let-polymorphism hence is the show-stopper for the unstaging translation. However attractive, we cannot use the translation for implementing MetaOCaml (unless we give up on polymorphic let within brackets, which is unpalatable). Therefore, MetaOCaml currently takes the steep implementation route: modifying the OCaml front-end to account for brackets and escapes, and the painful patching of the type-checker to implement the staged type system of Figure 1. After the type-checking, the staging constructs are eliminated by a variant of the unstaging translation [14]. That translation manipulates OCaml's Typedtree, which represents the AST after type-checking. Although the tree bears OCaml types, it is 'untyped': it is the ordinary data structure that does not enforce any typing or scoping invariants. Manipulating the tree is error-prone, with no (mechanically checked) assurances of correctness.

4 A New Translation of Quoted let-expressions

We now present the new translation for quoted let-expressions, which works even with polymorphic let-bindings. We attempt at the 'rational derivation' of the translation, with our constant refrain of copying vs. sharing.

The previous §3.2 showed a straightforward translation for quoted let-expressions, which converts

```
(1)  .<let x = 1::[] in
      (2:: x ,3:: x)>
```

(the quoted version of the first example of §2.1) to the following code-combinator based code

```
(2)  let_ (cons (int 1) nil ) @@ fun x →
     pair (cons (int 2) x) (cons (int 3) x)
```

This example does not have let-polymorphism. But if it did, we are in trouble: the x let-binding of (1) is converted to the x lambda-binding of (2). In the Hindley-Milner type-system lambda-bindings, unlike let-bindings, are not generalizable. We see the dead end, regardless of how the let_ combinator is implemented.

To have any hope of generalization, we need a translation that could map a let-binding in the quoted code to a let-binding. The putative translation should convert (1) into something like

```
(3)  comb1 (let x = comb2 (cons (int 1) nil) in
            comb3 (pair (cons (int 2) x) (cons (int 3) x)))
```

where comb1, comb2, and comb3 are yet to be determined combinators. This proposal seems to be the most general compositional, syntax-directed translation that has the desired let-binding. It fits within the unstaging translation of §3.1 in other ways: the future-stage variable x in (1) of the type int list is mapped in (3) to the present-stage variable of the expected (see Figure 4) type int list cod. After all, this the only type that makes, say, cons (int 2) x well-typed.

All is left is to appropriately implement comb1, comb2 and comb3, for all realizations of the Code interface. Proposition 3 imposes a constraint: Evaluating (3) with the CodeCode implementation should give back (1). And here we notice something odd. The expression (cons (int 1) nil) evaluates to .<[1]>., according to the existing code-combinators of CodeCode. The result of comb2 (cons (int 1) nil) should hence be or contain that singleton list; let us write it as .<. . . [1]. . . >.. The let-expression in (3) then produces comb3 .<(2::(. . . [1]. . .)),(3::(. . . [1]. . .))>.. Code-generating combinators may only combine pieces of code received as arguments but can never deconstruct or examine them. Therefore, it does not seem possible that our result can lead to (1), regardless of what comb1 or comb3 might do. We have already inlined .<[1]>., which we should have let-bound and shared instead.

The only way forward is to have comb1 .<[1]>. to somehow generate something like .<**let** y = [1] **in** body>. and return the let-bound variable as a code value, that is .<y>.. That does not seem possible either. To build code for a let-expression we need the code for the RHS of the binding, and the code for the body. The combinator comb1 does get the RHS code as the argument; but where is the body?

Fortunately we are stuck at the opportune place: the problem we are facing is real – but it has been solved long time ago in the partial-evaluation community. The solution is called 'let-insertion' [2, 17] and requires access to continuations. The delimcc library of OCaml [12] has exactly the control operators needed to implement the let-insertion interface[7]:

```
(4)  type α scope
     val new_scope: (ω scope → ω cod) → ω cod
     val genlet : ω scope → α cod → α cod
```

These combinators can be used as follows:

```
(5)  new_scope @@ fun p →
       lam (fun x→ add x (genlet p (add (int 1) (int 2))))
```

[7]This let-insertion interface is introduced here for the sake of translating quoted expressions and hence the pattern of use for genlet and new_scope is determined by the translation.

With the CodeCode implementation below it generates .<**let** y = 1+2 **in fun** x → x + y>., which *shares* the result of the subexpression 1+2 across all invocations of the function. In other words, genlet p e inserts, at the place marked by the corresponding new_scope, a **let** statement that binds e to a fresh variable, and returns the code with the name of that variable. We can finally complete the tentative translation (3):

```
(6)  new_scope @@ fun p →
       let x = genlet p (cons (int 1) nil ) in
       pair (cons (int 2) x) (cons (int 3) x)
```

With the CodeCode implementation of the combinators that expression indeed evaluates to (1).

Formally, the new translation of let-expressions takes the form

```
(7)  ⌈ let x = e1 in e2 ⌉ ↦
     new_scope (fun p → let x = genlet p ⌈ e1 ⌉ in ⌈ e2 ⌉)
```

Our running example with let-polymorphism, example (2) from §2.1 repeated below

```
(8)  .<let x = [] in (2::x,"3" ::x)>.
```

is hence translated to

```
(9)  new_scope @@ fun p →
     let x = genlet p nil in
     pair (cons (int 2) x)
          (cons (str "3") x)
```

which type-checks, and (with the CodeCode combinators) gives back (8). Incidentally, the combinator code without genlet

```
(10)  new_scope @@ fun p →
      let x = nil in
      pair (cons (int 2) x)
           (cons (str "3") x)
```

also type-checks. However, it generates

```
(11)  .< (2::[], "3" ::[]) >.
```

where [] is inlined rather than shared. The genlet combinator hence implements the sharing in the generated code rather than in the generator. The fact that the let-variable x in (9) gets the polymorphic type is the indication, and the vindication, of the equivalence of copying and sharing in this case. Although the RHS of the let-binding in (9) is an expression – moreover, an effectful expression, as we are about to see – the generalization happens anyway, thanks to the relaxed value restriction, recalled in §2.1. The type variable in α list cod occurs in the covariant position: note the covariance annotation +α cod in the Code signature.

The code that should not type check in MetaOCaml

```
(12)  .<let x = ref [] in (rset x 2, rset x "3")>. (* Does not type−check! *)
```

is translated to

```
(13)  new_scope @@ fun p →
      let x = genlet p (ref_ nil ) in
      pair (rset x (int 2))
           (rset x (str "3"))
      (* Does not type−check! *)
```

and is rejected by OCaml as expected: the type variable α in the inferred type α list **ref** cod for x is non-variant and is not generalized; x does not get the polymorphic type and hence cannot be used in the differently typed contexts.

We implemented genlet, directly based on [17], for all three realizations of the Code signature: not just for CodeString but also for CodeReal and CodeCode, to demonstrate soundness:

```
(14)  module CodeLetReal = struct
        include CodeReal
        open Delimcc  type α scope = α cod prompt
        let new_scope body = let p = new_prompt () in push_prompt p (fun () → body p)
        let genlet p e = shift0 p (fun k → let t = e () in k (fun () → t))
      end

(15)  module CodeLetString = struct
        include CodeString
        open Delimcc  type α scope = α cod prompt
        let new_scope body = ... the same
        let genlet p e =
         let tvar = gensym "t" in
         shift0 p (fun k → "let␣" ^ tvar ^ "␣=␣" ^ e ^ "␣in␣" ^ k tvar)
      end

(16)  module CodeLetCode = struct
        include CodeCode
        open Delimcc  type α scope = α code prompt
        let new_scope body = ... the same ...
        let genlet p e = shift0 p (fun k → .<let t = .~e in .~(k .<t>.)>.)
      end
```

(The code of new_scope is identical in all three implementations, although the realizations of the abstract type α scope differ.) The type α prompt and the delimited control operators push_prompt and shift0 are provided by the delimcc library [12].

The genlet is so powerful that it easily moves bound variables

```
(17)  new_scope @@ fun p →
        lam (fun x→ add x (genlet p (add x (int 2))))
```

resulting in the generated code **let** y = x + 2 **in fun** x → x + y with the unbound variable x. One may prevent such undesirable behavior either with a complex type system (whose glimpse can be caught in [11]) or with a dynamic test, as implemented in MetaOCaml [14]. In our case, however, genlet appears in the code solely as the result of the translation of a quoted expression. Fortunately, our translation of let-expressions puts new_scope “right above” genlet, never letting them be separated by a lam binding. In this case, delimited control, which underlies genlet, is safe (for proofs, see [10]).

4.1 Value Restriction at the Whole New Level

Alas, our new translation stumbles for the common case, of polymorphic function bindings such as the following:

```
(1)  .<let f = fun x → x in (f 2, f "3")>.
```

The translation

```
(2)  new_scope @@ fun p →
     let f = genlet p (lam (fun x → x)) in
        pair (app f (int 1)) (app f (str "3"))
     (* Does not type−check! *)
```

is rejected by OCaml: the genlet expression has the type $(\alpha{\to}\alpha)$ cod, which is not covariant in α. Generalizing expressions of such types is unsound[8]: otherwise, we will have to accept the following

[8]However, if the target language of code generation has no ‘dangerous’ effects and does not need value restriction, we may as well allow generalizing expressions of the type t cod regardless of the variance of type variables in t.

clearly undesirable code – the quoted version of our running villain, the bad example (10) of §2.1.

```
(3)   .<let f = let r = ref [] in
                fun x → rset r x
        in (f 1, f "3")>.
      (* Does not type−check! *)
```

whose translation

```
(4)   new_scope @@ fun p1 →
        let f = genlet p1
          (new_scope @@ fun p2 →
             let r = genlet p2 (ref_ nil) in
             lam (fun x → rset r x)) in
        pair (app f (int 1)) (app f (str "3"))
      (* Does not type−check! *)
```

would have type-checked had we allowed generalization for the genlet p1 expression.

The problematic staged code (3) does not type-check according to the system of Figure 1 (and in MetaOCaml): the (GenLet) rule does not apply because the RHS of the let-binding in (3) is not syntactically a value. Hence we need something like the value restriction to likewise prevent generalization in (4) while still allowing it in (2).

Therefore, we amend the translation of let-expressions, (7) in §4, with the following

```
(5)   ⌈ let x = fun z → e1 in e2 ⌉ ↦
      new_scope (fun p → let x = genletfun p (fun z → ⌈ e1 ⌉) in ⌈ e2 ⌉)
```

where

```
(6)   val genletfun : ω scope → (α cod → β cod) → (α→β) cod
      (* provisional ! *)
```

is a new code-combinator to be added to the let-insertion interface. In other words, our translation should recognize when a **let**-bound expression is syntactically a function, and use genletfun rather than the general genlet combinator.

With the amended translation, the good example (1) is translated as

```
(7)   new_scope @@ fun p →
      let f = genletfun p (fun x → x) in
      pair (app f (int 1))
           (app f (str "3"))
      (* See the refined version below! *)
```

and will type-check. The translation (4) of the bad example (3) will have to use genlet rather than genletfun since the RHS of the let-expression in (3) is not syntactically a function. As we said, (4) does not actually type-check.

We have thus separated the let-insertion combinators into the general genlet and the specific genletfun, which applies only to the translation of what looks like a function. (We need similar genletX for other polymorphic values of non-covariant types, which are rare.) For genlet, generalization occurs only for covariant type variables; for genletfun, the generalization should occur always.

There remains a question how to make the generalization to always occur for genletfun expressions like those in (7), short of modifying the OCaml compiler. Incidentally, even Obj.magic does not seem to help us with expressions that the relaxed value restriction cannot generalize: an application of Obj.magic is not syntactically a value. The answer is admittedly a hack; nevertheless, it gives us another standpoint, however awkward, to hear the refrain of copying and sharing. And it also works with the extant OCaml compiler.

Let us step back to look at the clearly flawed translation of (1)

```
(8)   let f = fun () → (lam (fun x → x)) in
      pair (app (f ()) (int 1))
           (app (f ()) (str "3"))
```

and contemplate what is wrong with it. On the upside, the translated expression (8) does type-check: f is bound to a thunk (syntactically a value) and its type is hence generalized through the ordinary value restriction. Since f is bound to a thunk we have to add explicit () applications at each place it is used. Evaluating (8) with the CodeString implementation of code combinators shows the generated code ((**fun** x2 → x2 1), (**fun** x1 → x1 "3")), with the inlined rather than shared identity function. We had rather the identity function be let-bound and shared. Having learned that genlet introduces let-bindings into the generated code, the next attempt at the translation of (1) is

```
(9)   new_scope @@ fun p →
      let f = fun () → genlet p (lam (fun x → x)) in
      pair (app (f ()) (int 1))
           (app (f ()) (str "3"))
```

It also type-checks, since f is still bound to a thunk. The generated code

```
(10)   let t2 = fun x1 → x1 in
       let t4 = fun x3 → x3 in
       ((t4 1), (t2 "3"))
```

is still unsatisfactory: we had rather the two applications in the pair used the same binding of the identity function. When f () in (9) is first evaluated, it generates a let-binding and returns the code with the bound variable. We want the second invocation of f () to return the code for the very same bound variable. In other words, we would like to memoize f. Memoization [19] indeed was meant to make copying behave like sharing.

The trick hence is introducing a thunk into the let-binding in the translation to get around the generalization problem and introducing memoization to restore the sharing destroyed by thunking. In effect, we do 'double memoization': using genlet to 'memoize' the identity function in the generated code and memoize the invocation of genlet at the present stage. Once this is understood, the rest is straightforward. To make the translation similar to (7), we combine genlet with the memoization into genletfun:

```
(11)   type ω funscope
       val new_funscope : (ω funscope → ω cod) → ω cod
       val genletfun : ω funscope → (α cod → β cod) → (α→β) cod
```

The final translation of (1) then reads:

```
(12)   new_funscope @@ fun p →
       let f = fun () → genletfun p (fun x → x) in
       pair (app (f ()) (int 1))
            (app (f ()) (str "3"))
```

Unlike (7), we had to replace the occurrence of f with f () – explicitly marking the type instantiation, so to speak. This complication is still possible to implement with the source-to-source translation (call-by-name let-binding of [18] would be really handy here).

The double-memoizing genletfun can be easily and generically implemented, with a small bit of magic

```
(13)   type afun = | AFun : (α → β) cod → afun
                   | ANone : afun
       type ω funscope = ω scope ∗ afun ref
       let new_funscope body = new_scope (fun p → body (p, ref ANone))
       let genletfun : ω funscope → (α cod → β cod) → (α→β) cod =
         fun (p,r) body →
         match !r with
```

```
      | ANone → let fn = lam body in
                let x = genlet p fn in
                r := AFun x; x
      | AFun x → Obj.magic x
```

The code uncannily resembles (4) of §2.3.

One may wonder if it would be better to add genletfun to the OCaml type-checker as an ad hoc, always-generalize case. The answer at present should be "no": genletfun is still unsound, in the edge case of (4) of §2.3 – the example that is also unsound in the present MetaOCaml. Here is this example again, for reference

```
(14)   .<let f = fun () → .~( lift (ref [])) in
          ( rset (f ()) 2, rset (f ()) "3")>.
```

Its translation

```
(15)   new_funscope @@ fun p →
         let f = fun () → genletfun p (fun _ → csp (ref [])) in
         pair ( rset (app (f ()) (csp ())) ( int 1))
              ( rset (app (f ()) (csp ())) (str "3"))
```

type-checks – and when run with CodeReal exhibits the same segmentation fault it does in the case of the corresponding MetaOCaml code.

It seems our unstaging translation is just as sound – or unsound – as MetaOCaml. Solving the soundness problem of MetaOCaml described in §2.3 will make, we conjecture, the unstaging translation fully sound as well. Much work lies ahead.

5 Conclusions

We have presented a new, typing-preserving translation from a higher-order typed staged language, with hygienic quotations and unquotations, to the language without quotations. Code-generation is accomplished through a library of code-generation combinators. Our translation is remarkably simpler than other unstaging translations: it is not type-directed and can be accomplished as a source-to-source transformation. Mainly, the translation works for polymorphic let: let-expressions within quotes are transformed to also let-expressions, hence preserving generalization. All throughout the presentation we emphasized deep connections, between polymorphism and sharing.

Our translation is already a viable method of implementing staged languages. Yet the theoretical work has just began. Yet another feature of our translation is 'bug-preservation': the restrictions and unsound edge cases of let-polymorphic expressions are preserved in the translation. The problems hence can be investigated in a simpler setting, without staging.

We thus propose a research program:

1. Formally establishing the equivalence properties of CodeReal, CodeCode and CodeString and formally justifying the translation;
2. Generalizing from two-stage to multiple-stages, that is, to multiple levels of quotations;
3. Proving that the edge case described in §2.3 is the only one where genletfun is unsound;
4. Relaxing the value restriction even more so that genletfun could be implemented without magic;
5. Investigating trade-offs of various solutions to the unsoundness problem in §2.3 and finding the solution with the least loss in expressiveness and convenience.

Acknowledgments

I thank Jacques Garrigue and Atsushi Igarashi for many helpful discussions. Comments and suggestions by Yukiyoshi Kameyama and anonymous reviewers are gratefully appreciated. This work was partially supported by JSPS KAKENHI Grant Numbers 22300005, 25540001, 15H02681.

References

[1] Henry G. Baker (1993): *Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same*. *ACM SIGPLAN OOPS Messenger* 4(4), pp. 2–27, doi:`10.1145/165593.165596`. Available at `http://home.pipeline.com/~hbaker1/ObjectIdentity.html`.

[2] Anders Bondorf (1992): *Improving Binding Times Without Explicit CPS-Conversion*. In: *Lisp & Functional Programming*, pp. 1–10, doi:`10.1145/141471.141483`.

[3] Matt Brown & Jens Palsberg (2016): *Breaking through the normalization barrier: a self-interpreter for f-omega*. In Rastislav Bodik & Rupak Majumdar, editors: *POPL '16: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, pp. 5–17, doi:`10.1145/2914770.2837623`.

[4] Cristiano Calcagno, Eugenio Moggi & Walid Taha (2004): *ML-Like Inference for Classifiers*. In: *ESOP*, *LNCS* 2986, pp. 79–93, doi:`10.1007/978-3-540-24725-8_7`.

[5] Luca Cardelli (1987): *Basic Polymorphic Typechecking*. *Science of Computer Programming* 8(2), pp. 147–172, doi:`10.1016/0167-6423(87)90019-0`.

[6] Chiyan Chen & Hongwei Xi (2005): *Meta-Programming Through Typeful Code Representation*. *Journal of Functional Programming* 15(6), pp. 797–835, doi:`10.1017/S0956796805005617`.

[7] Wontae Choi, Baris Aktemur, Kwangkeun Yi & Makoto Tatsuta (2011): *Static Analysis of Multi-staged Programs via Unstaging Translation*. In Thomas Ball & Mooly Sagiv, editors: *POPL '11: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, pp. 81–92, doi:`10.1145/1925844.1926397`.

[8] Jacques Garrigue (2004): *Relaxing the Value Restriction*. In: *FLOPS*, *LNCS* 2998, Springer-Verlag, Berlin, pp. 196–213, doi:`10.1007/978-3-540-24754-8_15`.

[9] Yukiyoshi Kameyama, Oleg Kiselyov & Chung-chieh Shan (2008): *Closing the Stage: From Staged Code to Typed Closures*. In: *PEPM*, pp. 147–157, doi:`10.1145/1328408.1328430`.

[10] Yukiyoshi Kameyama, Oleg Kiselyov & Chung-chieh Shan (2011): *Shifting the Stage: Staging with Delimited Control*. *Journal of Functional Programming* 21(6), pp. 617–662, doi:`10.1017/S0956796811000256`.

[11] Yukiyoshi Kameyama, Oleg Kiselyov & Chung-chieh Shan (2015): *Combinators for impure yet hygienic code generation*. *Science of Computer Programming* 112 (part 2), pp. 120–144, doi:`10.1016/j.scico.2015.08.007`.

[12] Oleg Kiselyov (2012): *Delimited control in OCaml, abstractly and concretely*. *Theor. Comp. Sci.* 435, pp. 56–76, doi:`10.1016/j.tcs.2012.02.025`.

[13] Oleg Kiselyov (2013): *MetaOCaml Lives On: Lessons from implementing a staged dialect of a functional language*. ACM SIGPLAN Workshop on ML.

[14] Oleg Kiselyov (2014): *The Design and Implementation of BER MetaOCaml - System Description*. In: *FLOPS*, *LNCS* 8475, Springer, pp. 86–102, doi:`10.1007/978-3-319-07151-0_6`.

[15] Megumi Kobayashi & Atsushi Igarashi (2015): *Polymorphic type system for the multi-stage calculus with references*. 32. Meeting of Japan Society for Software Science and Technology (in Japanese).

[16] Peter J. Landin (1966): *The Next 700 Programming Languages*. *Communications of the ACM* 9(3), pp. 157–166, doi:`10.1145/365230.365257`.

[17] Julia L. Lawall & Olivier Danvy (1994): *Continuation-Based Partial Evaluation*. In: *Lisp & Functional Programming*, pp. 227–238, doi:10.1145/182409.182483.

[18] Xavier Leroy (1993): *Polymorphism by name for references and continuations*. In: *POPL '93: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, pp. 220–231, doi:10.1145/158511.158632.

[19] Donald Michie (1968): *"Memo" Functions and Machine Learning*. *Nature* 218, pp. 19–22, doi:10.1038/218019a0.

[20] Robin Milner (1978): *A Theory of Type Polymorphism in Programming*. *Journal of Computer and System Sciences* 17, pp. 348–375. Available at http://www.diku.dk/undervisning/2006-2007/2006-2007_b2_246/milner78theory.pdf, doi:10.1016/0022-0000(78)90014-4.

[21] (2003): *POPL '03: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*.

[22] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller & Martin Odersky (2013): *Scala-Virtualized: linguistic reuse for deep embeddings*. *Higher-Order and Symbolic Computation* (September), doi:10.1007/s10990-013-9096-9.

[23] Tim Sheard & Simon L. Peyton Jones (2002): *Template Meta-programming for Haskell*. In Manuel M. T. Chakravarty, editor: *Haskell Workshop*, pp. 1–16, doi:10.1145/581690.581693.

[24] Walid Taha & Michael Florentin Nielsen (2003): *Environment Classifiers*. In POPL [21], pp. 26–37, doi:10.1145/640128.604134.

[25] Peter Thiemann (1999): *Combinators for Program Generation*. *Journal of Functional Programming* 9(5), pp. 483–525, doi:10.1017/S0956796899003469.

[26] Mads Tofte (1987): *Operational Semantics and Polymorphic Type Inference*. Ph.D. thesis, Edinburgh University. Laboratory for Foundations of Computer Science Technical Report ECS-LFCS-88-54.

[27] Andrew K. Wright (1995): *Simple Imperative Polymorphism*. *Lisp and Symbolic Computation* 8(4), pp. 343–355, doi:10.1007/BF01018828.

[28] Hongwei Xi, Chiyan Chen & Gang Chen (2003): *Guarded Recursive Datatype Constructors*. In POPL [21], pp. 224–235, doi:10.1145/640128.604150.